Static scheduling

There are many methods for dealing with the pipeline stalls caused by branch delay; we discuss four simple compile-time schemes in this subsection. In these four schemes, the actions for a branch are static—they are fixed for each branch during the entire execution.

Freezing the pipeline

The simplest scheme to handle branches is to *freeze* or *flush* the pipeline, holding or deleting any instructions after the branch until the branch destination is known. This method is very simple, both in hardware and software, but the branch penalty that comes from it is fixed and thus cannot be reduced by software.

Predicted-taken and -not-taken

A higher-performance and only slightly more complex scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed. This *predicted-not-taken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. If the branch is taken, however, we need to turn the fetched instruction into a `NOP` and restart the fetch at the target address.

An alternative scheme is to instead treat every branch as taken.

In either a predicted-taken or predicted-not-taken scheme, the compiler can improve performance by organising code so that the most frequent path matches the hardware's choice.

Delayed branch

A fourth scheme in use in some processors is called delayed branch. This technique was heavily used in early RISC processors and works reasonably well in the five-stage pipeline.

In a delayed branch, the execution cycle with a branch delay of one is:

```
branch instruction
sequential successor 1
```

```
branch target if taken
```

The sequential successor is in the *branch delay slot*. This instruction is executed whether or not the branch is taken.

Note

Although it is possible to have a branch delay longer than one, in practice almost all processors with delayed branch have a single instruction delay.

There are three ways the compiler can use to choose the delay slot instruction:

1. **From before**: the delay slot is scheduled with an independent instruction from before the branch. This is the best choice.
2. **From target**: the branch delay slot is scheduled from the target of the branch. This is preferred when the branch is taken with high probability, such as a loop branch.
3. **From fall-through**: the branch delay slot is scheduled from the not-taken fall-through.

The limitations of delayed branch scheduling arise from:

1. The restrictions on the instructions that are scheduled into the delay slots
2. Our ability to predict at compile time whether a branch is likely to be taken or not

To improve the ability of the compiler to fill branch delay slots, most processors with conditional branches have introduced a *cancelling* or *nullifying* branch. In a cancelling branch, the instruction includes the direction that the branch was predicted. When a branch behaves as predicted, the instruction in the branch delay slot is simply executed as it would normally be with a delayed branch. When the branch is incorrectly predicted instead, the instruction in the branch delay slot is simply turned into a `NOP`.